APMA 1940X: Information and Coding Theory
Project 1

Albert Caputo

February 21 2018

1 Shannon vs. Shakespeare

Claude Shannon's work on the entropy of English led to the discovery that English, despite its complexity, is for the most part a very redundant language. In other words, English relies on certain combinations of letters such to achieve the same meaning as a single letter, such as i before e or u after q. Moreover, English is not much different from randomized letters in that there is only slight less entropy compared to a uniform distribution over the alphabet. Fascinatingly, the entropy of Shakespeare's plays [1] is remarkably similar to modern English despite being over 400 years old!

We will first analyze the probability and entropy of N-Grams in Shakespeare's plays as a whole and compare them to modern English and Shannon's work. This will include conditional distributions. We will then examine the case of sampling letters conditional on the last n letters starting with a random n-tuple from the distribution of n-grams. Next, we will examine a string from Shakespeare's *The Merchant of Venice* without spaces and attempt to recover word breaks using the minima of mutual information. [2] Lastly, MATLAB code and its relevant explanations will be provided.

1.1 The Probability and Entropy of N-Grams

The probability of single letters in Shakespeare is very similar to that of English, albeit slightly more uniform. Let us consider the probability of the 26-letter (ignoring space) alphabet. Furthermore, let us compare these letter probabilities compared to those from the **Concise Oxford Dictionary** (9th edition, 1995). [3] This should give us a somewhat feasible, although biased, representation of how letter frequencies have evolved over the past 400 years.

[1] As from: shakespeare.mit.edu/

[2] All data will be rounded to nearest hundreth and log = log2

[3] https://www3.nd.edu/ busiforc/handouts/cryptography/letterfrequencies.html

Letter	Shakespeare Probability (%)	Oxford Probability (%)
'A'	7.68	8.50
'B'	1.59	2.07
'C'	2.29	4.54
'D'	3.93	3.38
'E'	11.88	11.16
'F'	2.11	1.81
'G'	1.78	2.47
'H'	6.25	3.00
'I'	6.70	7.54
'J'	0.12	0.20
'K'	0.93	1.10
'L'	4.47	5.49
'M'	2.91	3.01
'N'	6.44	6.65
'O'	8.26	7.16
'P'	1.50	3.17
'Q'	0.10	0.20
'R'	6.26	7.58
'S'	6.58	5.74
'T'	8.71	6.95
'U'	3.47	3.63
'V'	1.01	1.00
'W'	2.35	1.29
'X'	0.13	0.29
'Y'	2.48	1.78
'Z'	0.05	0.27

Some notable changes include: 'H' (-3.25%), 'C' (+2.25%), 'T' (-1.76%), 'P' (+1.67%), 'R' (+1.32%). Upon close inspection of Shakespearean text, these changes are somewhat apparent. Otherwise, letter probability seems to have not changed significantly.

Let us now consider the entropy of Shakespeare when compared to Shannon's findings. We will examine the entropy of the distribution of n-grams followed by the conditional entropy F_n, i.e. the entropy of the next letter given the last n letters such that

$$F_n = H_{n+1} - H_n$$

where

$$H_n = -\sum_{x \in \mathbb{A}^n} p_n(x) \log p_n(x)$$

and $\mathbb{A}^n$ is the alphabet of possible n-grams. For instance, $\mathbb{A}^1 = \{\text{“}A\text{”}, ..., \text{“}Z\text{”}\}$, $\mathbb{A}^2 = \{\text{“}AA\text{”}, \text{“}AB\text{”}, ..., \text{“}ZY\text{”}, \text{“}ZZ\text{”}\}$... A 27-letter alphabet would include

space. Mining through the text of all of Shakespeare's plays has led to the following conclusions:

Alphabet	H_0	H_1	H_2	H_3	H_4
26-letter	4.70	4.19	7.85	11.08	13.85
27-letter	4.75	4.09	7.44	10.18	12.41

These values give way to the following conditional entropies:

Alphabet	F_0	F_1	F_2	F_3
26-letter	4.19	3.66	3.23	2.77
27-letter	4.09	3.35	2.74	2.22

Note that $H_0 = -\log(\frac{1}{26})$ is the entropy of the uniform alphabet and that $F_0 = H_1$ (not $H_1 - H_0$). We see that entropy decreases by roughly 0.5 bits per letter given. Compare this to the values found by Shannon [4]:

Alphabet	F_0	F_1	F_2	F_{word}
26-letter	4.14	3.56	3.3	2.62
27-letter	4.03	3.32	3.1	2.14

In the case of Shannon, $F_{word} \approx F_3$ as his calculation is based on the entropy of 4.5 letters whereas F_3 is based on the entropy of 4 letters. Interestingly, there is very little difference between the values found by Shannon and those obtained through Shakespeare!

1.2 Generation of Text Using N-Gram Distributions

Using the distribution F_n, we can generate random text by first picking a random (n+1)-tuple of letters from the text and continuously sampling from the conditional distribution to find the next letter given the past n letters. As we sample from larger conditonal distributions, we will begin to see more structure in the language. It will also become more obvious that we are sampling from Shakespeare! To begin, let us sample from the single letter distribution, i.e. sample letters including space independent of any previous letters.

P_1 (First Order):

"ME F OESOL VBUAE N OENEN TSPSEMLICAHASY I SY U ORITTN A MUEEYUOYRENADOIEMRLKLI OYEWETHFA DORO STR EOOACOHNA SRU HAT KRD EAITN E IGYSHT EUTONEU ONSERSEKXJENERFHOWESPV ENCAONLN EHAOTCO S LESIHTEHW"

[4] As from with notation modification:
http://people.seas.harvard.edu/ jones/cscie129/papers/stanford_info_paper/entropy_of_english_9.htm

No structure here, although certain aspects of the distribution become apparent, such as the frequency of "E".
Let us now sample from the distribution $P_2(b|a)$ where we sample new letters given the last letter starting with two letters.

P_2 (Second Order):

"ORARAROFOWO HENGOMACOFAN PIDIALAN IADILLY SONG
WALOOULCAREDD
HIR BB IRNE PAD BE HE FRLIE AVELD OWALE S Y O
IDITENCHORALEANEREBE
GDEUTHASOUS SCE T CLPLL WHI TICHEIND F WIT WIND T
ASISHEATATITHETH THEY P"

We got lucky and generated a few words, although there still isn't much structure, nor is it apparent what we are sampling from. Let's now sample $P_3(c|ab)$.

P_3 (Third Order):

"ON OTHY THY LOODS TIO RAND USEETE BUTILL AND MAD BE
PENRY LANQUE
NOT WALADGE WIT TOWN MAN AT HEAR THLED COMMAY SY
IMEN DICH BESS
WASSAIN BES EAR TH GO YOU FALL BERS ARD TABOT SON SNE
THER BOTHIS A SUP TH"

Now we are beggining to see some structure! The text is also beggining to appear somewhat "Shakespearean." Lastly, let's sample from $P_4(d|abc)$.

P_4 (Fourth Order):

"MAKE TO A GOOD PURGED I DID MERESSIES QUAKE THIS NOW
ME THIS
COND PERCY THE MORE QUES SHALL QUENCHLESS THE WITH
QUINY BE IN THIS
DEPARONG PRINCE APON IN THE GOD SIR SIR HERS PLEANSWERE
DISSINE BEGGAR AS W"

We now see more structure than nonsense and the essence of Shakespeare is much more apparent! Of course we still have not conditioned enough to create any sense of meaning. Perhaps a good enough actor could pass this off as authentic...

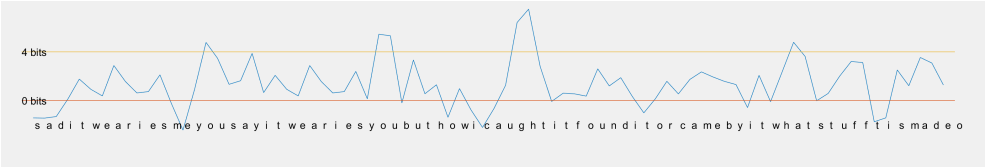
1.3 Using Mutual Information to Predict Word Breaks

If given a text with spaces removed, we can attempt to find word breaks using mutual information. In essence, we are measuring the "distance" of two probability distributions from being independent. If we find that there is very small

mutual information between, for instance, two sets of two letter strings, it is likely that there should be a space between them. The formula we use is as follows:

$$MI(ab, cd) = \log \frac{P_4(abcd)}{P_2(ab)P_2(cd)}$$

where ab is the leftmost pair of letters and cd is the rightmost pair of letters. Let us calculate the mutual information between pairs of two-letter strings for a line of dialogue from *The Merchant of Venice* and plot each value above the text and attempt to detect word breaks.



Certain words and letter combinations such as 'you', 'what', and 'augh' have very high mutual information, making it easier to detect the spaces before and after letters near them. Other aspects of this text are more difficult to determine where exactly the line break should go. In most cases, it is best to choose the lowest minimum amongst other local minima, as this should avoid putting spaces where they don't belong.

As expected, this algorithm will not always be correct due to its lack of complexity. However, it is impressive how such a simple algorithm is correct in finding word breaks more often than not.

1.4 MATLAB Code and Explanations

All of Shakespeare's plays combined yield about 5 megabytes of data. This can lead to computational time issues when processing the text. As such, let us describe a method to use base 26 or 27 encoding to minimize memory allocation.

Consider the case when the length of strings is 4. We are given an array of numbers such that each number corresponds to the letter that is indexed in the alphabet at that number (space = 27). Our first processing step is to create a rolling window matrix such that the ith row contains the i^{th} through $i + 3^{rd}$ letters, which visually looks like this:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

This is computationally inexpensive in MATLAB through using the hankel() function. In MATLAB,

```
function output = createRollingWindow(vector, n)
% CREATEROLLINGWINDOW returns successive overlapping windows onto a vector
%   OUTPUT = CREATEROLLINGWINDOW(VECTOR, N) takes a numerical vector VECTOR
%   and a positive integer scalar N. The result OUTPUT is an MxN matrix,
%   where M = length(VECTOR)-N+1. The I'th row of OUTPUT contains
%   VECTOR(I:I+N-1).
l = length(vector);
m = l - n + 1;
output = vector(hankel(1:m, m:l));
end
```

Next, we simply convert the rows of this matrix into a single integer base 26 or 27. Let's assume we are using space. Hence, for a string of length n, we multiply our rolling window minus 1 by $27^{[01\ldots n]^T}$. Using our example,

$$\left(\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} - 1\right) \begin{bmatrix} 1 \\ 27 \\ 27^2 \\ 27^3 \end{bmatrix} = M_4$$

where M_4 is a column vector containing each 4-letter string in the text represented as a number base 27. In MATLAB,

```
G4space = createRollingWindow(G-1,4)*[1;27;27^2;27^3];
```

where G is the array of each letter in number form.

Now we can easily compute the frequency, and thus probability, of each 4-string in the text without using too much computational effort using the MATLAB function tabulate(). Once we have our probability column vector P, we can achieve the entropy of the distribution:

$$H = -P^T \log P$$

Thus, in MATLAB,

```
function [entropy, distribution] = calcProb(vector)
% CALCPROB returns the tabulated distribution and entropy of the
% distributuion

% Tabulate - COL1 - COL2 - COL3 -
%          - Number - Occurences - Probability -
distribution = tabulate(vector);

% Eliminates 0log0
distribution = distribution(distribution(:,2) > 0,:);
```

```
% Convert probability distribution from percentage to [0,1]
dist = distribution(:,3)/100;

% Calculate entropy
entropy = -dist' * log2(dist);

end
```

Now that we have the probability of each 4-string, we need to convert back to characters. This is done via modular arithmetic. Our goal is to achieve a vector x such that each element represents the letter at the index position. Thus, given a single encoded integer N,

$$
\begin{aligned}
x_1 &= N \bmod 27 \\
x_2 &= \frac{(N - x_1) \bmod 27^2}{27} \\
x_3 &= \frac{(N - x_1 - x_2 \cdot 27) \bmod 27^3}{27^2} \\
x_4 &= \frac{(N - x_1 - x_2 \cdot 27 - x_3 \cdot 27^2) \bmod 27^4}{27^3}
\end{aligned}
$$

We then simply add 1 to x_i to achieve the i^{th} letter. Note that the above algorithm can be done iteratively for any string length. In MATLAB,

```
function output = decodeChar_(number,n,B)
%DECODE Converts base B encoding to string of letters of length n

% Initialize x and output
x = zeros(n,1)';
output = char(zeros(n,1)');
x(1) = mod(number,B);
output(1) = char('A'+x(1));

if n>1
        for i = 2:n
                % Computes the Base B decoded string
                x(i) = mod(number - x(1:i-1)*(B.^(0:i-2))',B^i)/B^(i-1);
                % Converts to char
                output(i) = char('A'+x(i));
        end
end

% Converts '[' to space via char decoding
output = strrep(char(output),'[',' ');
end
```

Now that we can decode each integer, let us store the probability of each string with it's associated key (in character form) in a hash map. This will make it easy to sample from conditional distributions. In MATLAB,

```
function dict = createHashMap(dist,n,B)
%CREATEHASHMAP Takes as input a distribution, the dimension n, and the base B, a
%outputs a hash map with n-length strings and their associated probability

% Initialize Hash Map
dict = containers.Map;

for i = 1:size(dist,1)
        % Set each key as decoded n-strings
        %     each value as associated probability
        dict(decodeChar_(dist(i,1),n,B)) = dist(i,3)/100;
end
end
```

Now we can sample from this dictionary to generate text! First, we sample a random 4 string from the hash map according to P. Next, we find the keys of the hash map that start with the last 3 letters of this string. To obtain the conditional distribution, we use the following MATLAB code,

```
function [output,strList] = condProb(dict,sample)
%CONDPROB is used to sample from P(letter|sample)

% Initialize list of all possible keys
str = string(keys(dict));

% Keep only those that start with sample
strList = str(startsWith(str,sample));

% Initialize probability vector
prob = zeros(size(strList,2));

for i = 1:size(strList,2)
        % Gather each probability from strList
        prob(i) = dict(char(strList(i)));
end

% Normalize for conditional probability
output = prob/sum(prob);
end
```

From there, we sample from the distribution of only 4 strings containing these keys and append the last letter. We do this 200 times to generate the text seen in section 1.2. Thus, in MATLAB,

```
function output = makeShakespeare(dict,n)
```

```
%Samples 200 characters from Shakespeare initial P4 distribution and
%continues using P4(d|abc)

% Extract String, Probability arrays
strList = string(keys(dict));
initialProb = cell2mat(values(dict));

% Sample from initial P4 probability distribution
init = strList(find(rand<cumsum(initialProb),1,'first'));
output = init;

for i = 1:200
        % Gather last three letters
        reader = char(output);
        reader = reader(end-(n-2):end);

        % Sample from conditional P4 distribution
        [newProb,newStrList] = condProb(dict,reader);
        newStr = newStrList(find(rand<cumsum(newProb),1,'first'));

        % Gather last letter
        newStr = char(newStr);
        newLetter = newStr(end);

        % Append letter to output
        output = output + string(newLetter);
end
end
```

To obtain the results from section 1.4, we simply follow the above steps but in base 26 to achieve our distributions and calculate mutual information iteratively. Thus, putting it all together,

```
% SHAKESPEARE.M: Calculating the Probability of Letter Strings

% Code from Pattern Theory:
% Converts .txt to ascii
label = fopen('shakespeare.txt');
[hk,count] = fread(label,'uchar');
fclose(label);
F = double(hk);
F(F<33) = 27;
F(F>64 & F<91) = F(F>64 & F<91) - 64;
F(F>96 & F<123) = F(F>96 & F<123) - 96;
F = F(F<28);
F2 = [0;F(1:(size(F)-1))]+F;
G = F(F2<54);
```

```
% ----------------------------------------------------------

% RANDOM TEXT GENERATION --- WITH SPACE

% Convert strings to base 27 integers
G1space = G - 1;
G2space = createRollingWindow(G-1,2)*[1;27];
G3space = createRollingWindow(G-1,3)*[1;27;27^2];
G4space = createRollingWindow(G-1,4)*[1;27;27^2;27^3];

% Calculate Distributions and Entropies
[entropy1space,dist1space] = calcProb(G1space);
[entropy2space,dist2space] = calcProb(G2space);
[entropy3space,dist3space] = calcProb(G3space);
[entropy4space,dist4space] = calcProb(G4space);

% Create Hash Maps for each string and their associated probability
dict1space = createHashMap(dist1space,1,27);
dict2space = createHashMap(dist2space,2,27);
dict3space = createHashMap(dist3space,3,27);
dict4space = createHashMap(dist4space,4,27);

%Shannon's Game in order from P1 to P4
randomText1 = makeShakespeare(dict1space,1)
randomText2 = makeShakespeare(dict2space,2)
randomText3 = makeShakespeare(dict3space,3)
randomText4 = makeShakespeare(dict4space,4)

% MUTUAL INFORMATION --- NO SPACE

% Eliminate Space
G = G(G<27);

% Convert strings to base 26 integers
G1 = G - 1;
G2 = createRollingWindow(G-1,2)*[1;26];
G3 = createRollingWindow(G-1,3)*[1;26;26^2];
G4 = createRollingWindow(G-1,4)*[1;26;26^2;26^3];

% Calculate Distributions and Entropies
[entropy1,dist1] = calcProb(G1);
[entropy2,dist2] = calcProb(G2);
[entropy3,dist3] = calcProb(G3);
[entropy4,dist4] = calcProb(G4);

% Create Hash Maps for each string and their associated probability
```

```
dict1 = createHashMap(dist1,1,26);
dict2 = createHashMap(dist2,2,26);
dict3 = createHashMap(dist3,3,26);
dict4 = createHashMap(dist4,4,26);

% Mutual Information of First 80 Characters
charArray(1:83) = upper(char(96+G(1+100:183)));

% Calculate mutual info for each consecutive pair
mi = zeros(80,1);
for i = 1:80
        mi(i) = log2(dict4(charArray(i:i+3))/...
        (dict2(charArray(i:i+1))*dict2(charArray(i+2:i+3))));
end

% Code from Pattern Thoery
% Code to make nice output
% Plotting Mutual Infromation above actual characters
hold off, plot(mi((1:80)));
axis([0 81 -20 20]); axis off, hold on
plot([0 81], [0 0]); plot([0 81], [4 4]);
text(0,0,'0 bits'); text(0, 4,'4 bits')
for i=3:83
        text(i-2+0.15,-2,lower(charArray(i)))
end

% Aggregate Entropies
allEntropies = [-log2(1/26) entropy1 entropy2 entropy3 entropy4;
-log2(1/27) entropy1space entropy2space entropy3space entropy4space];
allEntropyRates = [-log2(1/26) entropy1 entropy2-entropy1
entropy3-entropy2 entropy4-entropy3;
-log2(1/27) entropy1space entropy2space-entropy1space
 entropy3space-entropy2space entropy4space-entropy3space];
```

2 The Statistics of DNA Sequences

DNA is, at its core, a massively long strong containing 4 possible symbols: A,C,G, or T. Thus, we can analyze the probability distributions of these symbols over a smaller chunk of DNA to gain some insight into what's going on.

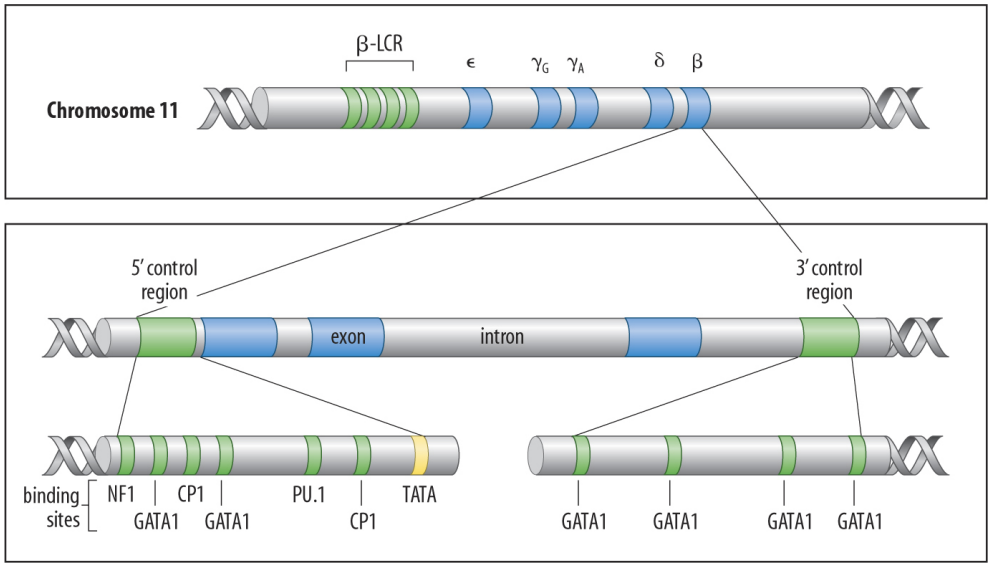
It is widely understood that smaller strings of length 3, called codons, correspond to units of genetic code. There are 64 possible codons and each have been tabulated by frequency in the human genome. What makes reading sequences of DNA so difficult is that nature gives no indication of where to start the sequence of triplets! Thus, there are three possible "phases," shifting the

break points by 0, 1, or 2.

Codons form together to create either exons or introns. Exons are known to be expressed code for amino acids which in turn form proteins. The end of a protein sequence is indicated by a STOP. Meanwhile, introns are believed to serve other purposes or are maybe junk. Research has led us to obtain the probability of each codon in all exons in the human genome. Thus, we can use this distribution to attempt to differentiate between introns and exons.

2.1 Getting the Data

We will be sampling from the DNA library GenBank [5]. In particular, we will examine a sequence of length 2000 from the well-studied beta globin sequence from base pairs 64001-66000. There are known to be three exons in this sequence, so we will do our best to find their starting position. A visual representation of the sequence we are trying to find is:



where the blue bars represent the three exons. Next, we need to acquire the probability distribution of each codon triplet for a Homo Sapien. This can be found via Kazusa[6], an online codon database. Now we are ready to analyze the sequence!

[5]Sample DNA from: https://www.ncbi.nlm.nih.gov/nuccore/U01317.1
[6]www.kazusa.or.jp/codon/cgi-bin/showcodon.cgi?species=9606

2.2 How to Find Exons

The main heuristic is that while codons in exons obey the probability distribution obtained from Kasuza, codons in introns do not. Thus, if we calculate the likelihood that a sequence is from this distribution versus the uniform distribution, we should have that exons will be achieve large positive values and introns will achieve small or negative values.

First, we will unwrap our sequence into a matrix such that each row is length 120 (i.e. 40 triplets) and each following row will be the previous row shifted by 3, introducing a new codon. The entire matrix will contain each subsequence of length 120 of our total sequence of length 2000. This process is nearly identical to that of section 1.4. We create one of these matrices for each possible break point so that we can analyze the likelihood of each of the three possibilities.

From here, we will calculate the following likelihood function for each subsequence starting at a:

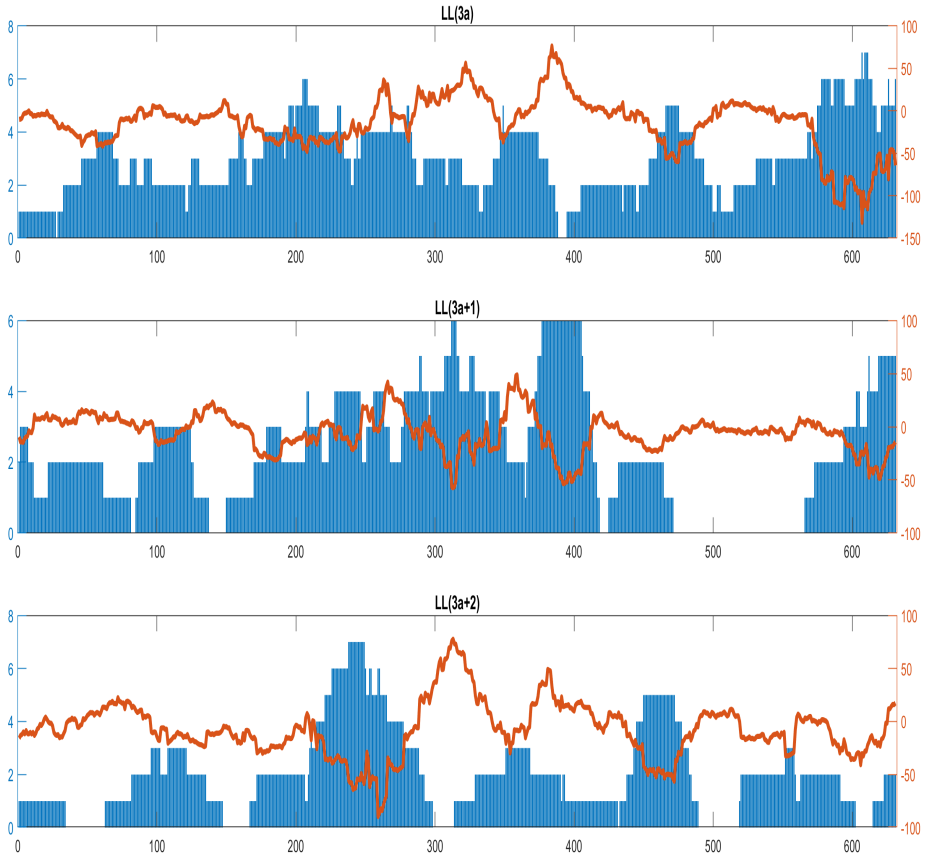
$$LL(a) = \sum_i n_i(a) \log(\frac{P_i}{U_i})$$

where $P = (P_1, ..., P_{64})$ are the human gene triplet probabilities, $U_i = \frac{1}{64}$ is the uniform distribution, and $n_i(a)$ is the number of occurrences of the ith triplet in the subsequence starting from a.

We will calculate and plot the values of $LL(3a), LL(3a+1)$, and $LL(3a+2)$ as a line plot. Furthermore, we will store the number of STOPs in each subsequence denoted by 'TGA','TAG', and 'TAA' and plot them as a bar on the same plot. We should see that, for one of these plots, there are broad peaks in the likelihood for each exon. Although this method will not give us the exact start of each exon, a STOP should indicate the end, which is where the bar plot will come in handy.

2.3 Plots and Analysis

The following plots are of the likelihood and number of stops per subsequence. The left axis represents number of stops while the right axis represents the likelihood value. The X-axis is the index of each subsequence.



It is immediately apparent that $LL(3a+2)$ contains the correct break point since there are reasonable spaces between STOPs and a few large peaks past the 300th subsequence. Of course, this does not immediately give away where the exons start and stop, but gives an idea where to look in our $LL(3a+2)$ vector.

Thus, our hypothesis is that the three exons exist around the 70th, 300th, and 380th sequences based on analysis of the plot. While the average exon encodes about 30-36 amino acids (90-108 bp long), exons could just encode a single amino acid [7]. This makes confirming our hypothesis tricky. We will take a look at subsequences between STOPs and attempt to extract the most likely sequence.

[7] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC99089/

Our first exon is believed to lie in the small, yet broad, peak toward the beginning of the plot. Since likelhood grows even after the first stop at around sequence 60, we should assume that the exon lies between this stop and the next stop at around sequence 80. Thus, we can extract the sequence that exon one most likely belongs to as follows:

'GTCTACTTGAAGAAGGAAAAACAGGGGGCATGGT'
'TTGACTGTCCTGTGAGCCCTTCTTCCCTGCCTCC'
'CCCACTCACAGTGACCCGGAATCTGCAGTGCTAG'

This sequence achieves a likelihood of 16.7 and ends with a 'TAG' stop. This lower likelihood is due to a smaller length sequence. Next, we examine the space where we hypothesize exon 2 might be.

Parsing through the likelihood vector $LL(3a+2)$, we have that there are no stops between subsequence 299 and 313. This indicates that the sequence in entirety should be about 165 base pairs long, ending in a stop. Using our likelihood formula, we can confirm that a sequence of this length has a likelihood value of 86.6, meaning that it is likely that an exon lies somewhere in this range. Although there is no guarantee that the entire sequence is the exon itself, we will assume it begins after the previous STOP. Thus, exon two is most likely contained in the sequence

'AGAAATCAAAAGAAGAAAATTCTAATATTCATGTTGCAGCCGTTTTTTGAATTTG'
'ATATGAGAAGCAAAGGCAACAAAAGGAAAAATAAAGAAGTGAGGCTACATCAAAC'
'TAAAAAATTTCCACACAAAAAAGAAAACAATGAACAAATGAAAGGTGAACCATGA'

ending with a 'TGA' STOP. Lastly, we examine where we think exon three might be.

This exon is trickier since we assume that it lies within the range of sequences to the peak nearest exon one, and there is a short subsequence containing two STOPs within this sequence. We will assume that the next exon does not lie within the subsequence and instead lies within the subsequence after. It is therefore believed that exon 2 lies in the sequence

'AAAGAAGAAAATCCTGCCATTTATGCGAGAATTGATGA'
'ACCTGGAGGATGTAAAACTAAGAAAAATAAGCCTGACA'
'CAAAAAGACAAATACTACACAACCTTGCTCATATGTGA'

which achieves a likelihood value of 17.5 and ends with a 'TGA' STOP. Again, the lower likelihood value is due to the smaller size of the sequence.

Although there are other possibilities for these exons, especially with exon one and two, these are the most likely arrangements according to the likelihood

function. More expertise in the field of genetics is required to determine the exact start and stops of these exons. However, even with such a robust and simple technique, we are able with certainty to determine the starting position mod 3.

2.4 MATLAB Code and Explanations

To begin, we needed to upload our DNA sample text of length 120. We do this in a manner identical to that of section 1 to remove unnecessary numbers and spaces to yield a single vector of character values. Next, we needed to create a rolling window to create a matrix of each subsequence of length 120. Again, we use nearly identical code as in section 1. Thus, in MATLAB,

```
function output = createRollingWindow(vector)
% CREATEROLLINGWINDOW modified for DNA problem
% Each row is every possible sequence of length 120

output = char(zeros(600,120));
for a = 0:627
output(a+1,1:120) = vector(3*a+1:3*a+120);
end
```

We create one matrix per shift, i.e. 0,1,2 so to identify each possible phase. Next, we needed to upload the probability distribution of each codon. This was done using readtable() on a CSV file containing these values. To ease the process of calculating likelihood, we created a hash map where each key is every possible triplet and each value is the corresponding probability.

The final step was to determine the likelihood for each possible sequence. To do this, we introduced a function that would parse through each row of each matrix and determine the number of occurrences of each triplet followed by the log likelihood. We do this by using a new hash map for counting and the same probability hash map from before as inputs to $LL(a)$. We will output the likelihood in column 1 as well as the number of stops in column 2 for plotting later. Thus, in MATLAB,

```
function LL = calcLL(M, prob)
%calcLL takes as input the matrix version of the DNA sequence and the
%probability of DNA codons to compute the likelihood for each subsequence.
%Additionally, we store the number of stops in LL column 2.

% Initiatialize LL
[rows, col] = size(M);
LL = zeros(rows, 2);

for j = 1:rows
        % Count number of codons per sequence
```

```
        countDict = containers.Map(keys(prob),zeros(64,1));
        for i = 1:3:col-2
                countDict(M(j,i:i+2)) = countDict(M(j,i:i+2)) + 1;
        end
        % Calculate likelihood ratio per sequence
        for i = 1:3:col-2
                LL(j,1) = LL(j,1) + ...
                countDict(M(j,i:i+2))*log2(64*prob(M(j,i:i+2)));
        end
        % Determine number of stops per subsequence
        LL(j,2) = LL(j,2) + countDict('TAA') + ...
         countDict('TAG') + countDict('TGA');
end
```

Lastly, we just plot the values we achieve as shown in section 2.3! Thus, putting everything together,

```
% Code from Pattern Theory:
% Converts .txt to ascii
label = fopen('DNA.txt');
[hk,count] = fread(label,'uchar');
fclose(label);
F = double(hk);
F(F<33) = 27;
F(F>64 & F<91) = F(F>64 & F<91) - 64;
F(F>96 & F<123) = F(F>96 & F<123) - 96;
F = F(F<28);
F2 = [0;F(1:(size(F)-1))]+F;
G = F(F2<54);
G = G(G<27);
G = upper(char(G + 96));

% M_k is the sequence G sorted by subsequences
% of length 120 starting at index k
M0 = createRollingWindow(G);
M1 = createRollingWindow(G(2:end));
M2 = createRollingWindow(G(3:end));

% Convert DNA sequence probability to table
table = readtable('DNAfrequency.csv');
table.Properties.VariableNames{1} = 'Triplet';
codons = table.Triplet;
prob = table.Probability;

% Initiate probability hash map
probDict = containers.Map;
```

```
% Key: Codon, Value: Probability
for i = 1:64
        probDict(replace(cell2mat(codons(i)),'U','T')) = prob(i);
end

% Calculate LL(3a), LL(3a+1), LL(3a+2)
LL0 = calcLL(M0,probDict);
LL1 = calcLL(M1,probDict);
LL2 = calcLL(M2,probDict);

% Create Plots for each Likelihood:
% Line plot represents likelihood
% Bar plot represents number of stops
ax1 = subplot(3,1,1);
yyaxis right
ylabel('Likelihood')
p1 = plot(ax1,LL0(:,1));
p1.LineWidth = 3;
yyaxis left
b1 = bar(LL0(:,2));
title('LL(3a)')

ax2 = subplot(3,1,2);
yyaxis right
p2 = plot(ax2,LL1(:,1));
p2.LineWidth = 3;
yyaxis left
b2 = bar(LL1(:,2));
title('LL(3a+1)')

ax3 = subplot(3,1,3);
yyaxis right
p3 = plot(ax3,LL2(:,1));
p3.LineWidth = 3;
yyaxis left
b3 = bar(LL2(:,2));
title('LL(3a+2)')
```